THE FILE COPY

UNLIMITED

DR109992 (2)

AD-A208 519

RSRE
MEMORANDUM No. 4268

ROYAL SIGNALS & RADAR ESTABLISHMENT

A PERSISTENT HEAP FOR ALGOL 68

Authors: S J Rees, G Cliff,
P D Hammond, N E Peeling

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

DTIC
ELECTE
JUN 06 1989
S H D

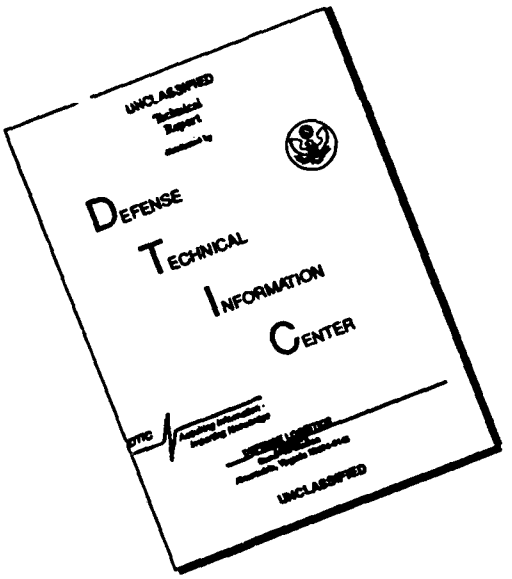
RSRE MEMORANDUM No. 4268

89 6 05 085

DISTRIBUTION STATEMENT A

UNLIMITED

DISCLAIMER NOTICE



THIS DOCUMENT IS BEST QUALITY AVAILABLE. THE COPY FURNISHED TO DTIC CONTAINED A SIGNIFICANT NUMBER OF PAGES WHICH DO NOT REPRODUCE LEGIBLY.

CONDITIONS OF RELEASE

0039124 BR-109992

********** U

COPYRIGHT (c)
1988
CONTROLLER
HMSO LONDON

********** Y

Reports quoted are not necessarily available to members of the public or to commercial organisations.

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4268

TITLE: A PERSISTENT HEAP FOR ALGOL68

AUTHORS: S.J. Rees G. Cliff P.D. Hammond N.E. Peeling

DATE: February 1989

SUMMARY

KeepSake[1] is a *multiuser data base kernel* which extends a programming language to enable simple data structures to be written to disc. It is not possible for the user to call KeepSake directly on compound data structures. This paper describes a method of extending the KeepSake procedures to facilitate the production of a persistent heap, in which all the data structures in the programming language can be written to disc.

Copyright
C
Controller HMSO London
1989

THIS PAGE IS LEFT BLANK INTENTIONALLY

A Persistent Heap for Algol68

Contents

DTIC COPY INSPECTED 6

Accession For
NTIS GRA&I ☑
DTIC TAB ☐
Unannounced ☐
Justification
By
Distribution/
Availability Codes
Dist | Avail and/or Special
A-1

THIS PAGE IS LEFT BLANK INTENTIONALLY

1.0 INTRODUCTION

KeepSake is a multiuser database kernel which extends a programming language to enable simple data structures to be written to disc. KeepSake write procedures take a block of data, write it to disc and return a pointer to that data. The read procedures take a KeepSake pointer and assign the data accessed to a user defined array. These procedures are used to construct a network of data and pointers. KeepSake comes with a sophisticated suite of data block management procedures, for example, disc garbage collection.

It is not possible for the user to call KeepSake directly on compound data structures; these must be broken into simple data structures before the KeepSake routines can be used. This memorandum describes how to extend KeepSake to allow a user to produce a persistent heap, i.e., to write a complete data type to disc, including any references, without having to unpack the data himself.

Although a KeepSake discpointer is a simple structure it is recognised as a "special" data type and is treated in a specific way because of the need to preserve the contents of the structure. It is therefore possible for the user to persist a data type which contains KeepSake discpointers to data which has already been persisted.

The software produced is in the form of two procedures, Persist and Unpersist. It was developed using Algol68 as the user's programming language on a VAX/VMS system; however the method used could be easily applied to other languages and machines.

The Persist routine requires information on the type of the object to be persisted and the object itself. Persist takes as its parameters a KeepSake discfile, a mainstore reference to the data structure which is to be persisted and a vector of characters which describes the mode of the data structure. Persist delivers a KeepSake discpointer as its result. Clearly it is inefficient to describe the data type to be persisted by a vector of characters as this requires the user to provide a block of text twice, once for the compiler and once for the Persist procedure, and discrepancies between the two are not detected. However, the Persist procedure needs to know the exact mode of the data type which is to be persisted, and has to accept the mode of any data structure no matter how complex, and this cannot be described without the use of an infinite union which is not implemented in languages such as Algol68 and C. An alternative approach would be to alter the compilers to build this in as is done already for read and print procedures. One language which has built in persistence is PS-Algol[2]; persistence in this language is totally transparent, there being only one pointer, the PNTR, for mainstore and filestore alike. In contrast to PS-Algol, Persist was designed to use specific pointers for filestore data so the user is always aware if he is addressing filestore or mainstore.

Unpersist has two parameters, the KeepSake discfile where the data was written and the KeepSake discpointer produced by Persist. It delivers as its result a mainstore reference to the data which had been persisted.

2.0 HOW TO USE PERSIST AND UNPERSIST

The procedure Persist takes as its first parameter a KeepSake DISCFILE. Directions on producing and initialising KeepSake discfiles are given in [1].

The second parameter of Persist is a VECTOR[]CHAR which gives all the declarations necessary to describe the mode of the object which is to be persisted. Each mode in the vector of characters must be described in terms of basic modes (INT, REAL etc) or of a mode which has preceded it in the character string. The syntax used to describe the modes is exactly that used in Algol68, which means that the user can use a text editor to extract the mode declarations from his program for use in the string. The final mode in the character string must be the mode of the object which is to be persisted. The following example may help:-

```
VECTOR[]CHAR id = "MODE  M1 = STRUCT(INT i,REAL r),"
                         "M2 = STRUCT(BOOL b, CHAR c),"
                         "M3 = STRUCT(M1 m, M2 n);";
```

The third parameter of Persist is an INTEGER which is the start address of the data to be persisted. This can be produced by using BIOP 99 to change the mode of a REF to an INTEGER as in the following example. Note especially that in the case of vectors and arrays a single pointer to the data is delivered by a REF REF MODE and this is the mode which must be changed to an INTEGER by BIOP 99.

The result of a successful call of Persist is a KeepSake DISCPTR. The user can write this DISCPTR away using the normal KeepSake routines or he can incorporate it into an Algol structure which can be the parameter of a subsequent call of Persist.

The following lines of code are all that need to be added to a user's program to enable him to persist a data structure of MODE EXAMPLE. The Persist procedure can only be used within a KeepSake environment and the KeepSake DISCFILE it uses must have been opened and initialised using standard KeepSake routines. The user must have access to the Algol68 module containing the Persist software.

```
DISCPTR discptr;

VECTOR[]CHAR id = "MODE   MYMODE = STRUCT(INT i, REF REAL x),"
                          "EXAMPLE = STRUCT(MYMODE m, INT n, REF[,]INT data);";

OP(REF EXAMPLE)INT RTI = BIOP 99;

EXAMPLE example := .....;

discptr := persist( keepsake_discfile, id, RTI(example));

        { "discptr" can be filed away using standard KeepSake
          routines to file DISCPTRs or it can be
          incorporated into an Algol structure and
          persisted later}
```

The procedure Unpersist takes as its parameters a KeepSake DISCFILE and DISCPTR. It delivers as its result an INTEGER which is the value of the mainstore pointer to the data type which was persisted. The user must convert this INTEGER to a REF MODE (or in the case of vectors and arrays a REF REF MODE) with a BIOP 99.

The data structure EXAMPLE persisted above can be unpersisted by adding the following code to a program. As with Persist the call of Unpersist must be within a KeepSake context and the KeepSake DISCFILE must have been opened and initialised using standard KeepSake routines. The KeepSake DISCPTR must have been delivered as the result of a call of Persist.

```
EXAMPLE recovered_data;
OP (INT) REF EXAMPLE ITR = BIOP 99;

recovered_data := ITR(unpersist(keepsake_discfile, discptr))

    {recovered_data can then be used normally }
```

3.0 DESCRIPTION OF THE METHOD USED BY PERSIST AND UNPERSIST

3.1 Persisting data

Briefly, Persist separates data items into mainstore addresses and literals, it repacks the data into a contiguous block and creates a separate table to show which elements of this data block are addresses. KeepSake discpointers are copied to a separate vector of discpointers and their locations in the data block are remembered in a second table. The data block, tables and discpointers are then filed away using basic KeepSake routines. It is necessary to tell KeepSake specifically of any discpointers included in the user's data to prevent their being lost during a KeepSake

garbage collection. A more detailed description of the Persist software is given below.

Persist uses a lexical reader and syntax analyser to convert the vector of characters which describe the data to be persisted into a format which it can use. Each data type is described simply as a number of bytes with pointers to those bytes which are KeepSake discpointers and mainstore addresses. The syntax analyser has to take account of the way the Algol68 compiler on VAX/VMS represents its data in mainstore.

The procedure which copies the data from mainstore to an output buffer is called recursively. At any call one data type is relocated as follows. Firstly all the bytes describing a data type, including references and KeepSake discpointers, are copied from mainstore and appended to an output buffer. At this stage any bytes in the data type which represent references will be incorrect and still contain the mainstore addresses. If the data type contains any KeepSake discpointers, copies of them are appended to a vector of discpointers and the position of bytes which represent discpointers in the output buffer is recorded in a separate vector. If the data type contains references, the procedure is called again to copy the data types pointed to by the references, and the bytes in the output buffer which represent the references are reset to be the element number of the output buffer where the first byte of the data they point to is written. As with discpointers the position of the reference bytes in the output buffer is recorded in a separate vector. This recursive method of relocating data deals with any data type including multiple linked lists. The following example illustrates how data is relocated.

Assume the data is to be filed as two vectors of integers. The first vector, V1 say, is the output buffer containing all the users data, literals and addresses. The literals will be correct but the addresses will be with respect to the beginning of the vector V1. The second vector, V2 say, is the reference table and will have one element for each reference in the users data, recording which elements of V1 are references.

Consider the following example :-

```
INT a:=1, b:=2, c:=3;
REAL d:=2.5;
MODE EXAMPLE = STRUCT(INT i, REF INT j, INT k, REF REAL l);
EXAMPLE s = (a, b, c, d);

The data type s would be written away as two vectors as follows :-

V1 = (1, 5, 3, 6, 2, 16672, 0)
V2 = (2, 4)
```

V2[1]=2 tells us that V1[2] is a REF. V1[2]=5 tells us that the data pointed to by the REF starts at V1[5]. As the REF was to an INTEGER the data occupies only one element of V1. However, consider V2[2]=4, this tells us that V1[4] is a REF;

V1[4]=6 tells us that the data pointed to by the REF starts at V1[6]. In this case the REF is to a REAL and the data occupies two elements of V1. (16672, 0) is the representation on Vax of 2.5 stored in two words.

Before a data type is relocated its reference is checked to see if it is a reference to NIL. This is a special case; no data is relocated and the bytes representing the reference are reset to the literal value NIL which for the Algol68 compiler on VAX/VMS is 0. In addition to checking for NIL the reference is checked to see if it is a reference to data which has already been relocated. This is achieved by comparing the reference and data length of the data type to be relocated with a table of these values for data already processed. There are three cases. In the simple case the data type is a complete copy of data already relocated; no further relocation is necessary and the bytes representing the reference are reset to be the element number of the output buffer where the first byte of the data they point to is already written. In the second case the data type to be relocated contains a data type which has been processed already, for example, a user persists an element of a vector and then the complete vector. In this case, after the data type has been relocated and its reference bytes reset, the original copy of the included data type is removed from the output buffer and its reference bytes are updated to its new position. The final case is when the data type to be relocated has some data in common with a previously processed data type, for example, the user persists two overlapping slices of a large vector. In this case the software relocates the union of the two data types and resets their reference bytes accordingly. Duplicated data is removed from the output buffer and the buffer is compressed. Data persisted in this way is consistent, in that should a data type contain addresses pointing to a common area of store, for example, a structure containing an array and a slice of the same array, only one copy of the array will be persisted and the slice will be a pointer into that array.

In practice data cannot be copied directly to an output buffer as, with large data types, this would cause storage problems. However, it is possible to record sufficient data to enable the buffers to be created, or partially created, at the time of data transfer.

3.2 Unpersisting data

Unpersist uses KeepSake routines to recover the filed data block, discpointers and tables. The discpointers in the data block are overwritten by the recovered discpointers using the locational information stored in the second table. The addresses in the data block are reset using the start address of the data block in mainstore and information from the first table. The mainstore start address is then delivered as the result. As an illustration of the method used by Unpersist, consider the data type EXAMPLE persisted above. The data was stored as two vectors as follows:-

```
V1 = (1, 5, 3, 6, 2, 16672, 0)
V2 = (2, 4)
```

Recovery of this data is simple. Both vectors, V1 the output buffer and V2 the reference table, are copied into mainstore and the mainstore address of the first element of V1 is established; say this is called final_address. Elements V1[2] and V1[4], which are known to be references from vector V2, are incremented by the value of final_address. All that now remains is to assign the value final_address to a variable of mode REF EXAMPLE. The above data type did not contain any discpointers. Had it done so, they would have been recovered, along with the table describing their location in the output buffer. The bytes representing discpointers in the output buffer would then have been overwritten by the recovered discpointers. It is necessary to reset the discpointers in the output buffer because they may have been given new values during a compacting garbage collection.

Storing the tables describing the data type on disc means that the mode of the object which has been persisted does not have to be given to Unpersist; an alternative solution would be to pass a vector of characters describing the persisted mode to Unpersist and to parse it to produce the tables.

4.0 TYPE CHECKING

Given access to the compiler, the "ideal" "safe" solution would be to file the compiler's description of the data type to be persisted with the persisted data. On recovery, this description could be compared with the compiler's description of the data type to which the recovered data will be assigned. This would ensure that data is never recovered into an incorrect data type. As Persist does not interact with the compiler, this is not possible and there is no check that the user has supplied the correct mode to Unpersist although, should data be recovered to an incorrect mode, it would almost certainly result in an access violation. The main concern is to ensure that data on disc is not corrupted by a user inadvertently recovering "rubbish" into a discpointer. Here we rely on KeepSake's routine type checking which will not accept discpointers which could not have been produced by KeepSake for the database in use. This means that an incorrect discpointer is almost certain to fail when it is submitted to a KeepSake read or write procedure.

5.0 ACKNOWLEDGEMENTS

The author has pleasure in thanking colleagues at RSRE, in particular Paul Hammond for all his advice and assistance during the development of this software. Also Gerard Cliff, a vacation student from Loughborough University, who developed the lexical reader and syntax analyser.

6.0 REFERENCES

[1] N.E. Peeling, K.R. Milner.
KeepSake : A Database Kernel.
RSRE Report No. 88014 December 1988.

[2] Ps-Algol Reference Manual 4th Edition.
PPRR-12.88. University of Glasgow & University of St Andrews.

THIS PAGE IS LEFT BLANK INTENTIONALLY

DOCUMENT CONTROL SHEET

Overall security classification of sheet .....UNCLASSIFIED.....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	Memo 4268		UNCLASSIFIED

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
7784000	ROYAL SIGNALS & RADAR ESTABLISHMENT ST ANDREWS ROAD, GREAT MALVERN, WORCESTERSHIRE WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

A PERSISTENT HEAP FOR ALGOL68

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
REES, S J	CLIFF, G	HAMMOND, P L	1989.02	6

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

UNLIMITED

Descriptors (or keywords)

continue on separate piece of paper

Abstract

KeepSake [1] is a multiuser data base kernel which extends a programming language to enable simple data structures to be stored on disc. This paper describes a method of extending the KeepSake procedures to facilitate the production of a persistent heap, in which all the data structures in the programming language can be stored on disc.

THIS PAGE IS LEFT BLANK INTENTIONALLY